INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Some Improvements on Multipartite Table Methods*

Florent de Dinechin, Arnaud Tisserand

**N° 4059**

Novembre 2000

THÈME 2


ISSN 0249-6399 ISRN INRIA/RR--4059--FR+ENG

RHÔNE-ALPES

# Some Improvements on Multipartite Table Methods

Florent de Dinechin, Arnaud Tisserand

Thème 2 — Génie logiciel
et calcul symbolique
Projet Arénaire

Rapport de recherche n° 4059 — Novembre 2000 — 16 pages

**Abstract:** This paper presents an unified view of most previous table-lookup-and-addition methods: bipartite tables, SBTM, STAM and multipartite method. This new definition allows a more accurate computation of the error entailed by these methods. Being more general, it also allows an exhaustive design space exploration which has been implemented, and leads to tables smaller than previously published ones by up to 50%. These tables have also been synthesised for Virtex FPGAs, and the paper discusses some results of this synthesis.

**Key-words:** Bipartite tables, table-lookup-and-addition methods, SBTM, STAM, multipartite, FPGA

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

# Quelques améliorations sur les méthodes à base de tables mutlipartites

**Résumé :** Ce papier présente une unification des travaux précédents sur les méthodes de calcul à base de tables et d'additions. Cette nouvelle vision du problème permet un calcul plus précis de l'erreur de ces méthodes. De part son caractère plus général, elle permet une exploration exhaustive de l'espace des paramètres, cette exploration a été implantée, et elle conduit à des tables jusqu'à 50% plus petites que celles précédemment publiées. Les tables obtenues ont été synthétisées sur des FPGA de type Virtex, les résultats de ces synthèses sont décrits dans ce papier.

**Mots-clés :** tables biparties, méthodes à base de tables et d'additions, SBTM, STAM, tables mutlipartites, FPGA

# 1 Introduction

Table-lookup-and-addition methods, such as the bipartite method, have been the subject of much recent attention [5, 2, 7, 8, 4]. They allow to compute commonly used functions with low accuracy (currently up to 24 bits) with a lower hardware cost than that of a straightforward table implementation, while being faster than shift-and-add algorithms à la CORDIC. They are particularly useful for providing initial seed values to iterative methods such as the Newton-Raphson algorithms for division and square root [3] which are commonly used in the floating-point units of current processors.

This paper clarifies some of the cost and accuracy questions which are incompletely formulated in previous papers. It also unifies two complimentary approaches to multipartite tables, by Stine and Schulte[8], and Muller[4]. This unified view completely defines the implementation space for multipartite tables, which in turn allows us to provide a methodology for selecting, in this space, the best implementation that full-fills arbitrary accuracy and cost requirements. This methodology has been implemented and is demonstrated on a few examples.

After some notations and definitions in section 2, section 3 presents previous table-and-addition methods, and unifies them as a general multipartite method. Section 4 shows how to explore the design space in order to select the best multipartite implementation full-filling a given accuracy requirement. Section 5 defines the values to be stored in the tables. Section 6 presents our implementation and exposes some results. Finally, Section 7 discusses these results and concludes.

# 2 Generalities

## 2.1 Notations

Throughout this paper, we discuss the implementation of a function with inputs and outputs in fixed-point format. We shall use the following notations.

- We note $f : [a, b[ \rightarrow [c, d[$ the function to be evaluated with its domain and range. The reader should keep in mind that all the following work can (and must) be straightforwardly extended to arbitrary closed, semi-closed or open intervals (the reciprocal, for example, is typically computed on $[1, 2[ \rightarrow ]0.5, 1]$). A general presentation would degrade readability without increasing the interest of the paper. Our implementation, however, allows such arbitrary combinations.
- We note $w_I$ the size (in bits) of the inputs to the implementation.
- We note $w_o$ the required output size, in bits.

In general, we will identify any word of $p$ bits to the integer in $\{0...2^p - 1\}$ it codes, and provide explicit functions to map such an integer into the domain or range of the function. For instance, if $x$ is an input word of $w_I$ bits, we also note $x$ the integer of $\{0...2^{w_I} - 1\}$ that it codes, and we map $x$ to the input interval $[a, b[$ by $\mu(x) = a + (b - a)x/2^{w_i}$.

## 2.2 Errors

Classically, we have three different kinds of error which affect to the global error of an evaluation of $f$:

- The input discretisation (or quantisation) error measures the fact that an input number usually represents a small interval of values centred around this number.
- The approximation error measures the difference between the pure mathematical function $f$ and the approximate mathematical function (here, a piecewise affine function) that will be used to evaluate it.
- Output discretisation (or rounding) errors measure the difference between the approximated mathematical function and the closest machine-representable value.

In the following we will ignore the question of input discretisation, by considering that an input number only represents itself as an exact mathematical number. A discussion about quantisation errors should come before or after the implementation presented here. Throughout the paper, we will point out the questions that depend on this hypothesis.

# 3 Table-and-addition methods

All the methods presented here consist in a piecewise affine approximation of the function.

## 3.1 The bipartite method

First presented by Das Sarma and Matula [5] in the specific case of the reciprocal function, and generalised by Schulte and Stine [7, 8] and Muller [4], this method consists in splitting the input word $x$ into three parts $x_1$, $x_2$, and $x_3$ of respective sizes $n_1$, $n_2$ and $n_3$ such that $n_1 + n_2 + n_3 = w_I$. Using the previous convention, the $x_i$ code integers in $0 \leq x_i < 2^{n_i}$ and we define $\mu(x)$ or $\mu(x_1, x_2, x_3)$ as the exact mathematical value corresponding to the input binary code $(x_1, x_2, x_3)$. For an input interval $[a, b[$ we have $\mu(x) = a + (b - a)2^{-n}x = a + (b - a)(2^{-n_1}x_1 + 2^{-n_1-n_2}x_2 + 2^{-n_1-n_2-n_3}x_3)$.

Fig. 1 shows an example function, with the three scales on the axis, for $n_1 = n_2 = n_3 = 2$: the axis is graduated in $x_1$ (largest dashes), $x_2$ (medium ones), and $x_3$ (smallest dashes).

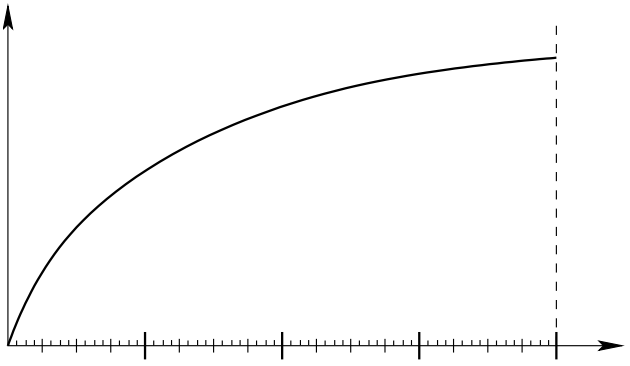

Figure 1: Graduating the input range

The idea is then to perform a first order Taylor approximation of $f$ at point $\mu(x_1, x_2, 0)$:

$$f(\mu(x_1, x_2, x_3)) = f(\mu(x_1, x_2, 0)) \; + \; \mu(0, 0, x_3) f'(\mu(x_1, x_2, 0)) \; + \; \varepsilon$$

A simple error analysis [4] shows that for $n_1 \approx n_2 \approx n_3$, it is possible to approximate the derivative $f'(\mu(x_1, x_2, 0))$ by $f'(\mu(x_1, 0, 0))$ and keep the total error in "acceptable bounds". Graphically, this means that the slope is considered constant on the bigger intervals (where $x_1$ is constant), as illustrated by Fig. 2, which is a zoom view on Fig. 1).

It is therefore possible to store in a first table indexed by $x_1$ and $x_2$ the value $f(\mu(x_1, x_2, 0))$, and in a second table indexed by $x_1$ and $x_3$ the offset computed for each value of the small interval, $\mu(0, 0, x_3) f'(\mu(x_1, 0, 0))$.

Obviously it is possible to build a better linear approximation than that of the Taylor formula, by slightly offsetting the value of the first table, and using a slope which is slightly different from the derivative at point $\mu(x_1, 0, 0)$. Fig. 3 shows such an approximation. Schulte and Stine [6] use for the first table the value of $f$ in the middle point of the smaller intervals, and for the slope the derivative in the middle point of the bigger interval. Another idea would be to use the average value of the slopes at both ends of the bigger interval. Section 4.3 will show how to compute exactly the slope and the offset that minimise the linearisation error.

In the following, we will thus no longer talk of Taylor approximation, but of a piecewise linear approximation. On each small interval, the two parameters of this approximation are:

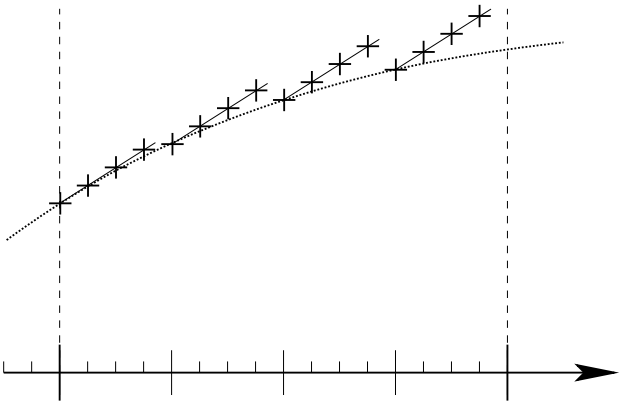

Figure 2: The bipartite approximation, using the Taylor first order expansion

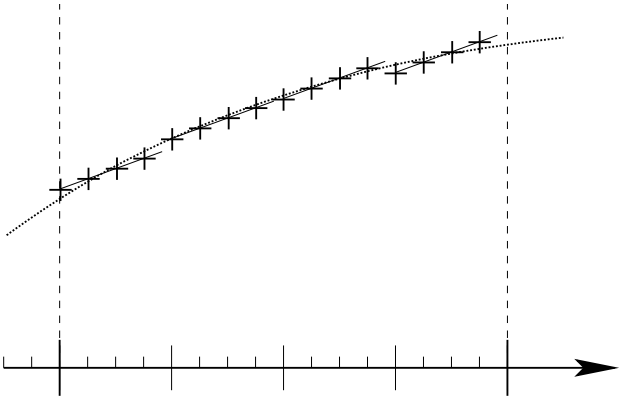

Figure 3: A better approximation

- an initial value stored in a first Table of Initial Values: TIV$(x_1, x_2)$,
- and a slope $s(x_1)$ which will be used to compute the offset to this initial value as a linear function of $x_3$, stored in a Table of Offsets TO$(x_1, x_3) = s(x_1)\mu(0, 0, x_3)$.

This will make it possible to perform more accurate estimations of the approximation error than by using the Taylor approximation formula: Taylor only gives an upper bound on the error, whereas we will be able to compute it exactly using the two parameters (slope and initial value) of the linear approximation on each interval. This will in turn allow a more accurate computation of the function.

## 3.2 The Symmetrical Bipartite Table Method (SBTM)

This is a small technical improvement in principle, but it leads to both an improvement in the approximation error and the size of the tables to implement. Schulte and Stine have remarked that it is possible to exploit the symmetry of the curve on each small interval (see Fig. 4). The first table now stores the value of the function in the middle of the small interval, and the second table stores the offset to this value, which due to symmetry needs only be stored on half the interval, and without its sign bit which is the same as that of the input word. The hardware cost of conditionally computing the opposites is more than compensated by the improvement in size of the second table [7].

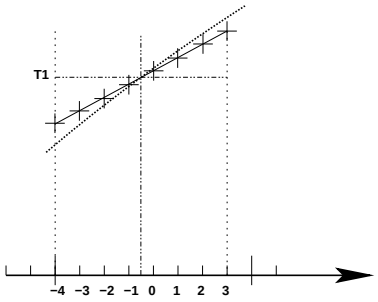


Figure 4: The SBTM method

### 3.3 The Symmetric Table and Addition Method (STAM)

In another paper [8], Schulte and Stine remarked that the second term of the Taylor approximation, $\mathrm{TO}(x_1, x_3) = s(x_1)\mu(0, 0, x_3)$, can be distributed over several tables by decomposing again the word $x_3$ into several sub-words $x'_3$, $x'_4$, ... $x'_n$. By noting $\mu_i(x'_i)$ the value of the sub-word $x'_i$ in the input range of $f$, we have $\mu(0, 0, x_3) = \mu_3(x'_3) + \mu_4(x'_4) + ... + \mu_n(x'_n)$, which leads to

$$s(x_1)\mu(0, 0, x_3) = s(x_1)\mu_3(x'_3) + s(x_1)\mu_4(x'_4) + ... + s(x_1)\mu_n(x'_n)$$

Thus we can replace the second table $\mathrm{TO}(x_1, x_3)$ with several tables $\mathrm{TO}_3(x_1, x'_3)$, ... $\mathrm{TO}_n(x_1, x'_n)$ which have fewer input bits, and a few additions. There will be two tradeoffs here:

- a cost tradeoff, between the cost of the additions and the table size reduction,
- an accuracy tradeoff: although the above equation is not an approximation, it will lead to more discretisation errors (one per table) which will sum up to a bigger discretisation error, unless the smaller tables have a bigger output accuracy (and thus are bigger). We will also formalise this tradeoff.

Note that for $j \neq i$ the weight of the LSB of $x'_i$ is not the same as the weight of an LSB of $x'_j$: their ratio is a power of two. This means that there is the same ration between the accuracies of $s(x_1)\mu_i(x'_i)$ and $s(x_1)\mu_j(x'_j)$. It is probably possible, therefore, to build even smaller tables than Schulte and Stine by compensating the higher accuracy by a rougher approximation on $s(x_1)$, obtained by removing some least significant bits from the input $x_1$. We will build up on this idea in 3.5.

### 3.4 Muller's Multipartite method

A paper from Muller [4] contemporary to that of Stine and Schulte indeed exploits this idea in a specific case. The multipartite method presented there is based on a decomposition of $x$ into $2p + 1$ sub-words of identical sizes $x_1$,...,$x_{2p+1}$, and a Taylor approximation at the point defined by the $p + 1$ first. The second term of this approximation is then distributed as previously, and an error analysis determines how many of the $p + 1$ first sub-words are needed to compute the derivative for each sub-term with sufficient accuracy.

It is found that equivalent accuracies are obtained by a table addressed by $x_{2p+1}$ and a slope determined only by $x_1$, a table addressed by $x_{2p}$ and a slope determined by $x_1$ and $x_2$, and in general a table addressed by $x_{2p+2-i}$ and the $i$ most significant sub-words.

Muller also shows that the error/cost tradeoffs of this approach are comparable to the STAM method (although without any numerical value). His error analysis, however, imposes a rigid decomposition: all the sub-words are of equal sizes. One of our purposes is to define a more general decomposition allowing better tradeoff exploration.

### 3.5 General multipartite tables

Investigating what is common in both previous methods leads to define a decomposition of the input word into sub-words that is intermediate between the STAM and the multipartite method (and generalizes both). We apologise for the change of notation, whose purpose is to avoid confusion with the previous methods.

- The input word is split into two sub-words $A$ and $B$ of respective sizes $\alpha$ and $\beta$ with $\alpha + \beta = w_I$ (see Fig. 5).
- The most significant sub-word $A$ addresses the TIV.
- The least significant sub-word $B$ will be used to address $m \geq 1$ TO(s) (the case $m = 1$ will be the bipartite case).
  - B will in turn be decomposed into m sub-words $B_0$,..., $B_{m-1}$, the least significant being $B_0$. In the bipartite case $m = 1$ we have $B_0 = B$.
  - A sub-word $B_i$ starts at position $p_i$ and consists of $\beta_i$ bits (see Fig. 6). Necessarily $p_0 = 0$ and $p_{i+1} = p_i + \beta_i$
  - The sub-word $B_i$ is used to address the $i$-th TO, noted $\text{TO}_i$. This TO is also addressed by a sub-word $A_i$ of length $\alpha_i$ of $A$
  - The maximum linearisation error entailed by $\text{TO}_i$ is a function of $(\alpha_i, p_i, \beta_i)$ which is easy to evaluate, as Section 4.3 will show.
  - The implementation of these $\text{TO}_i$ will exploit their symmetry, just as in the STAM method.
- Finally, to lighten the notations, we will denote $\mathcal{D} = \{\alpha, \beta, m, (\alpha_i, p_i, \beta_i)_{i=1...m}\}$ a decomposition of the input word.

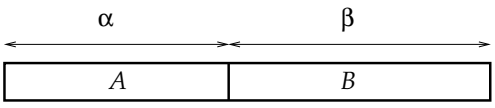


Figure 5: The input word...

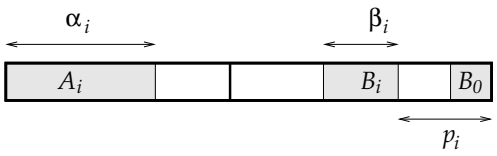


Figure 6: ...decomposed to address $\text{TO}_i$

Note that:

- the bipartite decomposition $n = n_1 + n_2 + n_3$ is a special case of our multipartite decomposition with $m = 1$, $\alpha = n_1 + n_2$, $\alpha_0 = n_1$, $\beta = \beta_0 = n_3$ and $p_0 = 0$.
- A STAM decomposition of $x$ into $x_1$, $x2$, ..., $x_n$ corresponds to $m = n - 2$, $A = x_1x_2$, $A_i = x_1$ for all $i$, and $B_0 = x_n$, ...,$B_{m-1} = x_3$.
- Similarly, Muller's multipartite tables are a specific case of our general decomposition where the $\alpha_i$ are multiples of the constant sub-word length.

It should be clear that general decompositions are more promising than Stine and Schulte's in that they allows to reduce the accuracy of the derivative part of the TOs (and thus their size). They are also more promising than Muller's, as they are more flexible (for example the size of the input word needs not be a multiple of some $2p + 1$). Our methodology will also be slightly more accurate than both in the error analysis. Section 6 will show the benefits of these improvements.

### 3.6 Hassler and Tagaki's method

In addition to all these methods based on Taylor/linear approximation, Hassler and Tagaki [2] have presented a method based on partial product arrays (PPAs). Stine and Schulte show in [8] that standard bipartite and multipartite methods are more area and time efficient, so we do not elaborate here further. The subject of PPAs is probably not closed, though, all the more as it is closely related to Distributed Arithmetic approaches of vector operations [1] which are still being actively researched.

## 4 Selecting the decomposition

This section shows how to navigate the space of possible multipartite implementations in order to select the best in term of speed or area that full-fills some accuracy requirements.

### 4.1 Error analysis

Like previous authors, we want to implement the function $f$ with faithful rounding: the computed result should be one of the two machine numbers closest to the (ideal) mathematical result. In other terms, the result should differ from the true result by less than one unit in the last place. Therefore we define the maximum output error as the value of the least significant bit (LSB) of the output: $\epsilon_f = (d-c)2^{-w_o}$ with the previous notations. We thus need to ensure that the total implementation error will be smaller than $\epsilon_f$. For this purpose we will need to compute with an internal precision which is higher than that of the final result: we will add $k$ bits to the tables to ensure this precision.

This error will then be the sum of three terms:

- a mathematical approximation error, whose maximum value will be noted $\epsilon_{\text{approx}}$ and will be computed exactly in 4.3,
- the rounding errors when filling each table, $\epsilon_{rt} \leq (m+1)\epsilon_t$ where $(m+1)$ is the number of tables and $\epsilon_t$ is the maximum rounding error when filling one table: $\epsilon_t = (d-c)2^{-w_o-k-1}$.
- the rounding error when rounding the sum of the tables to $w_o$ output bits: its maximum value is $\epsilon_{rf} = (d-c)2^{-w_o-1}$ in a straightforward implementation, but a trick due to Das Sarma and Matula [5] allows to improve it to $\epsilon_{rf} = (d-c)(2^{-w_o-1} - 2^{-w_o-k-1})$. This trick will be presented in section 5.

Finally, the condition to ensure faithful rounding, $\epsilon_{rt} + \epsilon_{rf} + \epsilon_{\text{approx}} < \epsilon_f$ is rewritten $k > -w_o - 1 + \log_2((d-c)m) - \log_2((d-c)2^{-w_o-1} - \epsilon_{\text{approx}})$.

Our methodology will be to set up formulas to compute exactly $\epsilon_{\text{approx}}$ as a function of the decomposition $\mathcal{D}$ of the input word. This will allow us to enumerate all the possible decompositions, to compute $\epsilon_{\text{approx}}$ for each of them, which will allow us either to reject a partition as unable to provide the required output accuracy, or to compute the size of the tables by computing $k$:

$$k = \left\lceil -w_o - 1 + \log_2 \frac{(d-c)m}{(d-c)2^{-w_o-1} - \epsilon_{\text{approx}}} \right\rceil$$

which in turn allows us to accurately evaluate the sizes and delays of the best candidates for implementation. These will then be synthesised.

### 4.2 An algorithm for choosing a decomposition

1. Fix a reasonable maximum for $m$. This will depend on the input and output accuracies, and on the performance expected, as a larger $m$ means more additions.
2. Enumerate the decompositions of an input sub-word. There is a large number of them, but each will need only a few computations.

3. For each decomposition $\mathcal{D}$, compute the approximation errors entailed by each $\mathrm{TO}_i$ as per 4.3, and sum them to get $\epsilon_{\mathrm{approx}}$. Keep only the decompositions for which this error is smaller than the maximum admissible error $\epsilon_f$. This gives a set of possible decompositions.
4. For possible decomposition, compute the number $k$ of extra accuracy bits, and use it to evaluate the size and speed of the implementation.
5. Synthesise the few best candidates to evaluate their speed and area accurately.

## 4.3 Actual computation of the approximation error

Here we consider a monotonic function with monotonic derivative (i.e. convex or concave) on its domain. This is not a very restrictive assumption: it is the case, after argument reduction, of all the functions studied by previous authors.

The error function we consider here is the difference $\varepsilon(x) = f(x) - \widetilde{f}(x)$ between the exact mathematical value and the approximation. Note that other error functions are possible, for example taking into account the input discretisation. The formulas set up in this section would not apply in that case, but it would be possible to set up equivalent formulas.

Our aim here is to exactly compute the contribution $\varepsilon_i^{\mathcal{D}}$ of $\mathrm{TO}_i$ to the total approximation error in the decomposition $\mathcal{D}$. For this purpose we will suppose that the rest of the tables (the TIV and the $\mathrm{TO}_j$ for $j > i$) provide infinitely accurate initial values.

We do not need to introduce symmetry considerations here: symmetry is a mathematical property of the linearisation that helps reduce the size of its implementation, but does not change the linearisation error (it does change the rounding error though).

Fig. 7 shows a sub-interval of $2^{w_I - \alpha_i}$ points of the input range: such an interval is indexed by $A_i$, and on it the slope of the linear approximations is constant: $s_i(A_i)$.

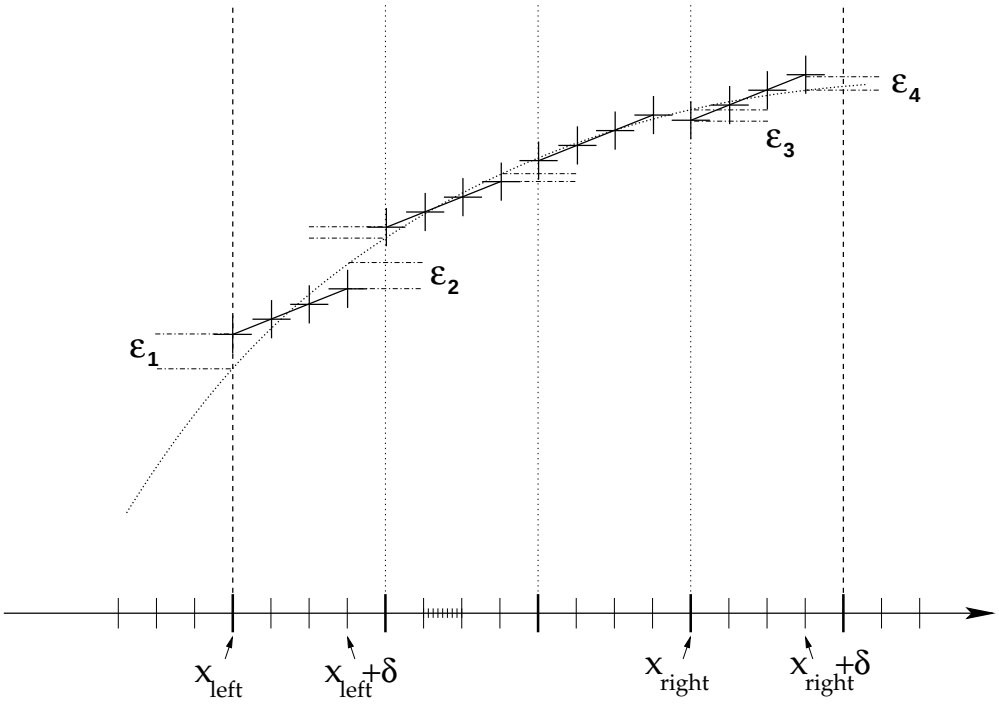


Figure 7: Computing the approximation error

This interval is itself split into intervals (delimited with bold dashes) where $\mathrm{TO}_i$ is in charge of implementing the linear approximation as shown on the figure: it will define the offsets for the $2^{\beta_i}$ points it

addresses. Note that there are in general $2^{\beta_0+\ldots+\beta_{j-1}} = 2^{p_i}$ points between each of the large dashes on the $x$ axis: other $\text{TO}_j$ for $j < i$ will be in charge for completing the approximation for these points, using sub-words $B_j$ and $A_j$.

So the function computed by $\text{TO}_i$ on a point of this interval is (before rounding) $\text{TO}_i(A_i, B_i) = s_i(A_i) \times \mu_i(B_i)$ where $\mu_i(B_i)$ is the value of the sub-word $B_i$ in the input range of the function $f$, that is:

$$\mu_i(B_i) = a + (b-a)2^{-w_I+p_i}B_i. \tag{1}$$

Fig. 7 shows the linearisation error on a few input points. It should be clear that, because of the convexity and the fact that all the slopes are equal on this interval, the maximum error is obtained on the borders of the interval, and more specifically when $\varepsilon_1 = -\varepsilon_2 = -\varepsilon_3 = \varepsilon_4$. We define the input values $x_{i\ \text{left}}^{\mathcal{D}}(A_i)$, $x_{i\ \text{right}}^{\mathcal{D}}(A_i)$ and $\delta_i^{\mathcal{D}}$ (or $x_{\text{left}}(A_i)$, $x_{\text{right}}(Ai)$ and $\delta$ to lighten the notations) as on the figure:

$$\begin{aligned}
x_{\text{left}}(A_i) &= a + (b-a)2^{-\alpha_i}A_i & (2)\\
x_{\text{right}}(A_i) &= a + (b-a)(2^{-\alpha_i}(A_i+1) - 2^{-w_I+p_i+\beta_i}) & (3)\\
\delta &= \mu_i(2^{\beta_i} - 1) & (4)
\end{aligned}$$

such that the maximum errors are obtained on $x_{\text{left}}$, $x_{\text{left}} + \delta$, $x_{\text{right}}$ and $x_{\text{right}} + \delta$:

$$\begin{aligned}
\varepsilon_1 &= \widetilde{f}(x_{\text{left}}) - f(x_{\text{left}})\\
\varepsilon_2 &= \widetilde{f}(x_{\text{left}} + \delta) - f(x_{\text{left}} + \delta)\\
&= \widetilde{f}(x_{\text{left}}) + s_i(A_i)\delta - f(x_{\text{left}} + \delta)\\
\varepsilon_3 &= \widetilde{f}(x_{\text{right}}) - f(x_{\text{right}})\\
\varepsilon_4 &= \widetilde{f}(x_{\text{right}} + \delta) - f(x_{\text{right}} + \delta)\\
&= \widetilde{f}(x_{\text{right}}) + s_i(A_i)\delta - f(x_{\text{right}} + \delta)
\end{aligned}$$

Remark that in the previous equations, $\widetilde{f}(x_{\text{left}})$ and $\widetilde{f}(x_{\text{right}})$ are the values to be computed by the TIV and the $\text{TO}_j$ for $j > i$.

In the system of three equations $\varepsilon_1 = -\varepsilon_2 = -\varepsilon_3 = \varepsilon_4$, the unknown are the slope $s_i(A_i)$ (on which the error obviously depends) and the values $\widetilde{f}(x_{\text{left}})$ and $\widetilde{f}(x_{\text{right}})$. Solving this system thus not only gives the approximation error, it also gives the value of the slope that minimises the error on each larger sub-interval, and the optimal initial values for $x_{\text{left}}$ and $x_{\text{right}}$. We get:

$$\begin{aligned}
\epsilon_i^{\mathcal{D}}(A_i) &= \frac{f(x_{\text{left}} + \delta) - f(x_{\text{left}}) - f(x_{\text{right}} + \delta) + f(x_{\text{right}})}{4} & (5)\\
s_i^{\mathcal{D}}(A_i) &= \frac{f(x_{\text{left}} + \delta) - f(x_{\text{left}}) + f(x_{\text{right}} + \delta) - f(x_{\text{right}})}{2\delta} & (6)\\
\widetilde{f}(x_{\text{left}}) &= f(x_{\text{left}}) + \epsilon_i^{\mathcal{D}}(A_i) & (7)\\
\widetilde{f}(x_{\text{right}}) &= f(x_{\text{right}}) - \epsilon_i^{\mathcal{D}}(A_i) & (8)
\end{aligned}$$

The last two equations can be generalised to give the initial value optimal for this $\text{TO}_i$ for each $B_i$. This can be exploited in the bipartite case, as we will do later in 5.1.

As a side note, remark that the slope that minimises the error is equal, in a second-order approximation, to the average value of the derivatives at points $x_{\text{left}}$ and $x_{\text{right}}$.

Now this error depends on $A_i$, but for the same argument of convexity, it will be maximum either for $A_i = 0$ or for $A_i = 2^{\alpha_i} - 1$: finally, the maximum approximation error due to $\text{TO}_i$ in the decomposition $\mathcal{D}$ is:

$$\epsilon_i^{\mathcal{D}} = \max(|\epsilon_i^{\mathcal{D}}(0)|, |\epsilon_i^{\mathcal{D}}(2^{\alpha_i} - 1)|) \tag{9}$$

and the total approximation error of the decomposition $\mathcal{D}$ is:

$$\epsilon^{\mathcal{D}}_{\text{approx}} = \sum_{i=0}^{m-1} \epsilon^{\mathcal{D}}_i \tag{10}$$

In practice, it is easy to compute this approximation error by implementing equations (9), (5), (4), (3), (2) and (1). Altogether it represents a few floating-point operations per decomposition.

## 4.4 The costs of a decomposition's implementation

Evaluating as exactly as possible the size and speed of the implementation of a multipartite decomposition is rather technology dependent, and is out of the scope of the paper.

We can, however compute exactly (as other authors) the number of bits to store in each table. This is the purpose of this section.

The actual costs (area and delay) of implementations of these tables and of multi-operand adder are the subject of current investigation. Section 6 will present some results for Virtex FPGAs, showing that the bit counts presented below allows a predictive enough evaluation of the actual costs.

### 4.4.1 The TIV

Once the number $k$ of additional bits has been determined, the size (in bits) of a TIV is simply $2^{\alpha} \times (w_o + k)$.

### 4.4.2 The TO$_i$

Now it is visible on the previous figures that the TO$_i$ have a smaller range than the TIV. More precisely, the range of TO$_i(A_i, *)$ is exactly equal to $|s_i(A_i) \times \mu_i(2^{\beta_i} - 1)|$. Again for convexity reasons, this range is maximum either on $A_i = 0$ or $A_i = 2^{\alpha_i} - 1$:

$$r_i = \max(|s_i(0) \times \mu_i(2^{\beta_i} - 1)|, |s_i(2^{\alpha_i} - 1) \times \mu_i(2^{\beta_i} - 1)|) \tag{11}$$

Once the number $k$ of additional bits has been determined (at step 4 of 4.2), the number of output bits of TO$_i$ is therefore

$$w_o(i) = \lceil w_o + k + \log_2(r_i/(d - c)) \rceil \tag{12}$$

In a symmetrical implementation of the TO$_i$, the size in bits of the corresponding table will be $2^{\alpha_i + \beta_i - 1} \times (w_o(i) - 1)$.

# 5 Filling the tables

## 5.1 The bipartite case

In this case we have only the TIV and one TO. We have seen that the value to fill in the TO can be chosen to minimise the approximation error, which in this case is the only approximation error. We can fill the TIV with the same purpose: now on each smaller interval we have a maximum and a minimum error which are on the borders (see Fig. 7). By storing the mean of the function at both borders of the interval, $(f(x) + f(x + \delta))/2$, we ensure that these maximum errors are equal in absolute value, which minimises the approximation error on the small interval. This is (very slightly) better than the approximation of Stine and Schulte, who take $f((x + \delta)/2)$.

### 5.2 The multipartite case

In the symmetrical multipartite case the previous minimisation is no longer possible because there are several $\mathrm{TO}_i$ involved. We therefore (similarly to Stine and Schulte, but more accurately) fill the TIV with the values of $f$ at points which will be the center of the intervals addressed by the $\mathrm{TO}_i$:

$$\mathrm{TIV}(A) = f\Big(a + (b-a)(2^{-\alpha}A + 2^{-w_I}(2^{p_1}(2^{\beta_1}-1)/2 + 2^{p_2}(2^{\beta_2}-1)/2 + ... + 2^0(2^{\beta_m}-1)/2)\Big)$$

### 5.3 Rounding the $\mathrm{TO}_i$

This section reformulates the techniques employed by Stine and Schulte in [8] and using an idea that seems to appear first in papers by Das Sarma and Matula [5].

First remark that there are two ways of rounding a real number to $w_o + k$ bits with an error smaller than $\epsilon_t = 2^{-w_o-k-1}$: the natural way is to round the number to the nearest $(w_o+k)$-bit number. Another method is to truncate the number to $w_o + k$ bits, and assume an implicit 1 in the $(w_o + k + 1)$-th position.

Before filling a $\mathrm{TO}_i$, remind that we will need to be able to compute the opposite of the value given by this table. In two's compliment, this opposite is the bitwise negation of the value, plus a 1 at the LSB. This leads us to use the second rounding method for the $\mathrm{TO}_i$: knowing that the LSB (here at position $w_o + k + 1$) is an implicit 1 means that its negation is a 0, and therefore that the LSB of the opposite is also an 1. There is therefore no need to add the sign bit at the LSB of the final adder, it is sufficient to store and bitwise negate the $w_o + k$ bits, and assume in all cases an implicit 1 at the $(w_o + k + 1)$-th position.

Now the TIV must be rounded in such a way to ensure that the (implicit) $(w_o + k + 1)$-th bit of the sum will always be a 1: this reduces the final rounding error from $\epsilon_{rf} = 2^{-w_o-1}$ to $\epsilon_{rf} = 2^{-w_o-1} - 2^{-w_o-k-1}$. Therefore, if $m$ is odd the first rounding method is used, if $m$ is even the second method is used.

Note that we don't need to store these implicit ones, but we must take them into account in the computation of the sum. The easiest way to do that is to add their sum to each value of the TIV at the position $(w_o + k)$. More precisely $m/2$ if $m$ is even, and $(m-1)/2$ if $m$ is odd. Thus a multi-operand adder taking the $m+1$ values of size $w_o + k$ adds all the implicit bits but one, and its result, extended by the remaining implicit bit, is the $(w_o + k + 1)$-bit number to round to $w_o$ bits.

### 5.4 Rounding the TIV

The final rounding consists in rounding a sum on $(w_o + k)$ bits, with one implicit one on the $(w_o + k + 1)$-th bit, to the nearest number on $w_o$ bits. This can be done by simply truncating the sum, provided we have added half an LSB of the final result to the TIV when filling it.

## 6 Implementation and results

### 6.1 Implementation

The methodology presented above has been implemented in a C++ program. This program performs the decompositions enumeration, chooses the best one with respect to accuracy and size, computes the actual values of the tables and finally generates a dedicated synthetisable VHDL file (only for bipartite at the moment).

In most of the previous works on the subject, the actual computation of the table values is missing or imperfectly described. Our tool therefore generates all the table values for the best decomposition. It allows to check that the final error is really smaller than the expected accuracy. We think that this information is useful and its determination is straightforward as soon as the tables are filled.

The ability to actually fill the tables also helps to caracterise the real quality of the final approximation. For instance, we can point some small problems such as the non-nonotonicities: the signs of the errors at the end of a segment and at the beginning of the next segment are opposite (see for instance Fig. 7 at the end of the third segment, for $x = x_{right}$). Therefore it is possible to have small non-monotonicities (no bigger

than one LSB thanks to faithful rounding) as illustrated. We have never seen any mention to this problem in the literature.

## 6.2 Multipartite results

Tables 1 and 2 present the best decomposition obtained for 16-bits and 24-bits operands for a few functions. In these tables, we compare with the best known results from the work of Schulte and Stine [8].

| $f$ | $m$ | $\alpha$ | $\beta$ | $\alpha_i$ | $\beta_i$ | tables | size | ref size |
|---|---|---|---|---|---|---|---|---|
| sin | 2 | 8 | 8 | 7,4 | 3,5 | $2^8 \times 20, 2^9 \times 11, 2^8 \times 9$ | 13056 | 20480 |
| | 3 | 6 | 10 | 6,6,4 | 3,3,4 | $2^6 \times 20, 2^8 \times 13, 2^8 \times 10, 2^7 \times 8$ | 8192 | 17920 |
| | 4 | 6 | 10 | 6,6,5,4 | 3,2,3,2 | $2^6 \times 20, 2^8 \times 13, 2^7 \times 10, 2^7 \times 8, 2^5 \times 5$ | 7072 | na |
| $2^x$ | 2 | 8 | 8 | 7,5 | 3,5 | $2^8 \times 20, 2^9 \times 12, 2^9 \times 9$ | 15872 | 14592 |
| | 3 | 6 | 10 | 6,6,4 | 4,3,3 | $2^6 \times 19, 2^9 \times 13, 2^8 \times 9, 2^6 \times 6$ | 10560 | 13568 |
| | 4 | 6 | 10 | 6,6,6,5 | 3,2,3,2 | $2^6 \times 21, 2^8 \times 15, 2^7 \times 12, 2^8 \times 10, 2^6 \times 7$ | 9728 | na |

Table 1: Best decomposition charateristics and table sizes for 16-bit operands

| $f$ | $m$ | $\alpha$ | $\beta$ | $\alpha_i$ | $\beta_i$ | tables | size | ref size |
|---|---|---|---|---|---|---|---|---|
| sin | 2 | 13 | 11 | 10,7 | 4,7 | $2^{13} \times 28, 2^{13} \times 15, 2^{13} \times 11$ | 442368 | 753664 |
| | 3 | 9 | 15 | 9,9,6 | 5,4,6 | $2^9 \times 29, 2^{13} \times 20, 2^{12} \times 15, 2^{11} \times 11$ | 262656 | 610304 |
| $2^x$ | 2 | 13 | 11 | 11,7 | 4,7 | $2^{13} \times 28, 2^{14} \times 15, 2^{13} \times 11$ | 565248 | 581632 |
| | 3 | 10 | 14 | 10,10,6 | 4,4,6 | $2^{10} \times 29, 2^{13} \times 19, 2^{13} \times 15, 2^{11} \times 11$ | 330752 | 425984 |

Table 2: Best decomposition charateristics and table sizes for 24-bit operands

The time required to obtained this decomposition results is very small in front of the time required for the synthesis. For instance, for 16-bit operands, it only take a few minutes to find the best decomposition for $m = 4$ multipartite table. For 24-bit operand, the decomposition exhaustive enumration can takes longer time. On a 400 MHz SUN Ultra 5 computer, it takes around 1 hour. In the future, we will provide guidelines to limit the exploration space. Indeed, it is possible to bound the minimal and maximal interesting values for some parameters such as $\alpha$, $\beta$ and the $\alpha_i$'s.

## 6.3 First FPGA implementation results

In this section, we present the very first results of the synthesis of the generated tables. Due to timing problems, we only have the complete results for the bipartite tables.

The target architecture is the Virtex device family from Xilinx. More precisely, we use a XCV400 FPGA with a speed grade of -4 (the slowest one). A 10% speed improvement has been obtained using a better speed grade (-5). The synthetised operator is considered as a combinatorial block. No pipelining is performed in this work (it is a future work). All operators have been synthesized using Leonardo Spectrum, the place and route operations are performed with Xilinx tools driven directly by Leonardo. For each architecture, the synthesis is performed with two goals: best effort on area and best effort on delay. Most of the time, the best architecture is the one obtained using the area constraint, but in a few cases, the delay constraint gives the fastest and the smallest architecture. The best synthesis constraints are reported. The metrics is the number of LUT (look up tables) which corresponds the basic cells of the FPGA.

### 6.3.1 Single tables

Table 3 and 4 present the implementation of single tables. These results show that the number of bits / number of LUTs ratio is more or less constant. This fact can be used to predict the size after synthesis on the FPGA from the table size in bits. From these tables we can deduce that the synthetiser perform some optimisation inside the table, because each LUT in a Virtex FPGA can only store 16 bits of memory (cf Xilinx documentation). We think that common small subwords are shared over close words. The low level optimisation of tables values will be one of our future work in this field.

| table | (6,10) | (6,11) | (7,12) | (8,14) | (8,15) | (9,16) | (10,17) | (10,18) | (11,19) |
|---|---|---|---|---|---|---|---|---|---|
| #BIT | 640 | 704 | 1536 | 3584 | 3840 | 8192 | 17408 | 18432 | 38912 |
| #LUT | 37 | 41 | 87 | 208 | 230 | 466 | 953 | 1022 | 2055 |
| #BIT/#LUT | 17.3 | 17.2 | 17.7 | 17.2 | 16.7 | 17.6 | 18.3 | 18.0 | 18.8 |
| delay (ns) | 5 | 5 | 6 | 10 | 12 | 11 | 12 | 13 | 16 |
| synth | 4″ | 4″ | 3″ | 44″ | 50″ | 3′54″ | 5′35″ | 6′17″ | 27′30″ |
| #LUT | 40 | 44 | 108 | 252 | 270 | 576 | 1241 | 1314 | 2773 |
| #BIT/#LUT | 16.0 | 16.0 | 14.2 | 14.2 | 14.2 | 14.2 | 14.0 | 14.0 | 14.0 |
| delay (ns) | 4 | 4 | 6 | 8 | 8 | 10 | 11 | 11 | 13 |
| synth | 9″ | 11″ | 20″ | 1′02″ | 1′04″ | 6′25″ | 9′48″ | 10′13″ | 52′39″ |

Table 3: Virtex FPGA implementation of single tables (the first table of bipartite table)

| table | (5,5) | (7,6) | (7,7) | (9,8) | (11,9) |
|---|---|---|---|---|---|
| #BIT | 160 | 768 | 896 | 4096 | 18432 |
| #LUT | 9 | 43 | 52 | 219 | 1098 |
| #BIT/#LUT | 17.8 | 17.9 | 17.2 | 18.7 | 16.7 |
| delay (ns) | 3 | 9 | 9 | 12 | 16 |
| synth | 1″ | 4″ | 10″ | 54″ | 7′57″ |

Table 4: Virtex FPGA implementation of single tables (the second table of bipartite table)

### 6.3.2 Bipartite tables

Tables 5 and 5 presents the complete implementation results for bipartite tables. The first table presents the results of the standard bipartite table solution without symmetry as proposed initially by Das Sarma and Matula in [5]. The second table presents the bipartite table solution with symmetry as presented by Shulte and Stine with the SBTM solution in [7] but with our slope and offset computations. If we compare with the values obtained with the SBTM, we have a small area improvment up to 7% less area. From these two tables, we can see that the symmetry slighlty increase the delay of the operator but leads to smaller tables.

# 7 Conclusion

We have presented several contributions to table-and-additions methods. The first one is to unify and generalise two complimentary approaches to multipartite tables, by Stine and Schulte, and Muller. The second one is to give a method of optimising such bipartite or multipartite tables which is more accurate than what could be previously found in the literature. Both these improvements have been implemented in

| $(W_I, W_O)$ | (8,8) | (10,10) | (12,12) | (14,14) | (16,16) |
|---|---|---|---|---|---|
| $(n_1, n_2, n_3)$ | (3,3,2) | (4,3,3) | (5,3,4) | (5,5,4) | (6,5,5) |
| $(o_1, o_2)$ | (9,4) | (11,5) | (14,7) | (16,7) | (18,8) |
| #BIT | 704 | 2048 | 7168 | 19968 | 53248 |
| #LUT | 51 | 131 | 409 | 1075 | 2898 |
| #BIT/#LUT | 13.8 | 15.6 | 17.5 | 18.6 | 18.4 |
| delay (ns) | 8 | 12 | 15 | 16 | 27 |
| synth | 8″ | 18″ | 1′29″ | 5′48″ | 30′24″ |

Table 5: Virtex FPGA implementation of bipartite tables without symmetry

| $(W_I, W_O)$ | (8,8) | (10,10) | (12,12) | (14,14) | (16,16) |
|---|---|---|---|---|---|
| $(n_1, n_2, n_3)$ | (3,3,2) | (4,3,3) | (5,3,4) | (5,5,4) | (6,5,5) |
| $(o_1, o_2)$ | (9,3) | (11,4) | (14,6) | (16,6) | (18,7) |
| #BIT | 624 | 1664 | 5120 | 17920 | 44032 |
| #LUT | 48 | 115 | 317 | 983 | 2308 |
| #BIT/#LUT | 13.0 | 14.5 | 16.2 | 18.2 | 19.1 |
| delay (ns) | 8 | 12 | 18 | 19 | 22 |
| synth | 8″ | 15″ | 56″ | 4′49″ | 25′32″ |

Table 6: VIrtex FPGA implementation of bipartite tables with symmetry

a general tool that can generate optimal multipartite tables from a wide range of specifications (input and output accuracy, delay, area). This tool outputs VHDL which has been synthesised for Virtex FPGAs.

Future work include completing the tool by allowing more accurate, technology-dependent area and speed estimations, taking into accound non-monotonicities, refining our algorithms to cut in the decomposition space in order to reduce the exploration time, and investigating possible low-level optimisations in the synthesis of the tables.

# References


[1] D. Benyamin, W. Luk, and J. Villasenor. Optimizing FPGA-based vector product designs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, April 1999.

[2] H. Hassler and N. Tagaki. Function evaluation by table look-up and addition. In S. Knowles and W.H. McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 10–16, Bath, UK, 1995. IEEE Computer Society Press, Los Alamitos, CA.

[3] J.M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[4] J.M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3):279–288, 1999.

[5] D. Das Sarma and D.W. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W. H. McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 17–28, Bath, UK, 1995. IEEE Computer Society Press, Los Alamitos, CA.

[6] M. Schulte and J. Stine. Symmetric bipartite tables for accurate function approximation. In T. Lang, J.M. Muller, and N. Takagi, editors, *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 175–183. IEEE Computer Society Press, 1997.

[7] M. Schulte and J. Stine. Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8):842–847, August 1999.

[8] J. Stine and M. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21(2):167, 1999.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399